Wordgames

Srini, rs@toprllc.com

February 15, 2022

1 Introduction

In this projectlet we will add to our list of games. The goal however is to be deliberate about each step, outlining the design constraints, considerations and where appropriate discuss some code fragments.

1.1 The game

Wordgames are very popular these days. My own destination every morning is the **the spelling bee** (`https://www.nytimes.com/puzzles/spelling-bee`) followed by **the letter boxed** (`https://www.nytimes.com/puzzles/letter-boxed`). Other games I frequent include **how strong is your vocabulary** (`https://www.merriam-webster.com/word-games/vocabulary-quiz`) and **wordle** (`https://www.powerlanguage.co.uk/wordle/`).

As a software engineer, these games intrigue me. How do you come up with a new game? Are these composed manually or automatically by a piece of software? Or even better can a program be written to solve these puzzles. The answer of course is **yes**. After all there are programs that solve **chess** problems, play **go**. So this projectlet is an attempt to build ourselves our very own game.

1.1.1 The model

The popular board game **Scrabble** will serve as our model. The key elements of this game are:

- There are a bank of tiles each with a letter on its face. There are also 2 tiles with a blank face.
- The tiles also have a score associated with them. For example a tile might have a face of A and a score of 1 while another with a face of X and a score of 10.
- The bag contains one tile of certain letters like X while several of other letters like A.

- Each player draws a certain number of tiles to always have 7 tiles on their side of the table and the player can form words with those tiles.

There are other important elements of Scrabble which we omit since the above are the ones we focus on in this projectlet.

1.1.2 Adaptation

We adapt the above key elements in subtle ways to make it amenable for our game. The significant adaptations are:

- This game will be a **solitary** one. You play with yourself.
- Once a set of tiles are drawn, you play till you quit with the same set of tiles.
- Words are formed with any subset of the tiles you draw. A letter can be used any number of times in a word.
- Each word is first verified against a dictionary. True words result in a word score. The games score is the sum of the scores of successful word scores.
- Guesses which turn out not to be dictionary words will result in a penalty of the total score.

1.1.3 Dealer vs Player

The projectlet gets to be a lot more interesting when we let the user deal the tiles where the software performs the **playing** function. In the player mode then, the software has to:

- Search for words that can be formed.
- Enumerate as many words as possible.

1.2 Deployment

The essential game can be deployed in several modes:

- A text based simple implementation is pretty straightforward and can be used to validate the game itself.
- The next step is to convert the game to a client server model. In this approach, a client app that can in turn play as the dealer or a player has to be produced. The interaction protocol then is a bespoke protocol.
- Scaling the deployment will use a web browser as the front end. The protocols in this case are much more standardized. The major benefit of this approach being the addition of a graphical interface using one of the standard frameworks.

2 Design

2.1 The Dictionary

https://github.com/redbo/scrabble/blob/master/dictionary.txt is a key resource - the list of words acceptable in a **scrabble** game. It is not clear if this is official but will suffice for our game. However our projectlet conceptually encompasses other dictionaries - a medical or scientific or literature oriented thus adding excitement to the game.

2.2 Scrabblet Core

https://www.thewordfinder.com/scrabble-point-values.php gives us a table indicating the count and scores associated with each letter in the **Scrabble** bank. In addition to letters, there are also blank tiles which have no score. The table translates easily into a representation as follows:

```
type tile is
   record
      face : character ;
      count : integer ;
      score : Integer ;
   end record ;

blank : constant tile := ( ' ' , 2 , 0 ) ;
subtype Faces is Character range 'A' .. 'Z' ;
bank : array(Faces'range) of tile := (others => blank) ;
```

A bag can be initialized thus:

```
 bank ('D') := ('D', 4, 2);
 bank ('E') := ('E', 12, 1);
 bank ('F') := ('F', 2, 4);
 bank ('G') := ('G', 3, 2);
 bank ('H') := ('H', 2, 4);
```

2.2.1 The Bag

Drawing of tiles from the bank requires that higher the count of a face the higher the chance of drawing that letter. A random drawing will result in **A** with 9 times the likelihood of **X** of which there is only one tile in the bank. In order to make this easy, we create a data structure called **bag** thus:

```
type Bag is array (integer range <>) of tile ;
type bagptr is access Bag ;
```

The essential idea is that we create an entry per tile resulting in for example 9 tiles for **A** but only one entry for **X**. Of course we should also have 2 entries for the blanks. Such an array can then be populated as follows:

```
    result := new Bag (1 .. tilecount);
    for f in bank'range loop
       for i in 1..bank(f).count
       loop
          bagidx := bagidx + 1;
          result(bagidx) := ( face => f , score => bank(f).score ,
                                          count => 0 ) ;
       end loop ;
    end loop ;
```

In the above fragment, for each character, we create entry each for the count. In this table the count does not serve any purpose so we set it to 0. We could have come up with a different structure without a count field as an alternative.

2.2.2 Random Selection

Once the bag is created as above, drawing a tile at random becomes straightforward - based on the idea that each tile is equally probable. Thus we can generate a uniformly distributed random number in the range 0.0 to 1.0 and depending on the result, a tile can be selected. We have to remember that a tile has been picked in order to avoid exaggerating the probability of each tile.

If we want to be really correct, every tile removal results in a larger probability that the remaining tiles have to be assigned In the interest of simplicity, this complication is not addressed in the example:

```
    while ridx < wl loop
       next := GNAT.Random_Numbers.Random (GEN) ;
       nidx := Integer(float( b'length ) * next)  ;
       if not taken(nidx)
```

```
          then
             ridx := ridx + 1 ;
             result(ridx) := b(nidx).face ;
             taken(nidx) := true ;
          end if ;
       end loop ;
```

2.2.3 Enumerate potential words

In our game, the above method is used to **deal** a set of tiles or letters to the player. Now the player can pick different subsets to form potential words. Such a subset let us call this **the hand** then can be used to lookup a dictionary to confirm that it is a word indeed and then compute a score to assign to the player.

It is conceivable that the hand just does not contain any dictionary words. In order to keep the player interested, the dealer should attempt to avoid words which dont have potential words. In this implementation the projectlet will:

- generate all subsets of letters starting with a length of 1 trying the length of the hand.
- compute the number of potential words formable with this subset
- generate a new hand if the number of words is less than a threshold.

As an illustration, supposing the hand turns out to contain the word **CRPEI** - the scrabble dictionary may yield the words:

CREEPIE	CREEPIER	CREPIER	CRIPE
EPEIRIC	PIECER	PIERCE	PIERCER
PRECIPE	PRECIPICE	PREPRICE	PRICE
PRICER	PRICIER	RECIPE	REPRICE

The hand is not a bad hand after all. The player can score quite a few points with this.

2.2.4 Anagrams

In order to enable the enumeration as stated above, and make the process reasonably performant, we adapt the notion of **anagrams** to include repetition of letters. In the above example there is only one **E** in the hand but words such as **PRECIPICE** and **CREEPIER** can be formed by using **E** any number of times. We achieve this by assigning a signature to each word in such a way that all the above letters have the same signature and thus are **anagrams** by our definition.

```
  Signatures : array (letter'range) of Positive
     := (
          2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
          59, 61, 67, 71, 73, 79, 83, 89, 97 , 101 );

  counted : array(letter'range) of boolean := (others => false);
  for word w
  for widx in w'range
  loop
     pragma Debug(Show(w(widx))) ;
     if not counted(To_Upper(w(widx)))
     then
        counted(To_Upper(w(widx))) := true ;
        result := result *
                  Interfaces.Unsigned_64(Signatures(To_Upper(w(widx))));
     end if ;
  end loop ;
```

In the above, each letter has a signature namely a **prime** number. By ensuring each letter is used only once, the computed signature becomes identical for all potential words of a given **hand**.

2.2.5 Words Database

We need an efficient data structure to hold all the anagrams. Predefined libraries of most programming languages support notions of containers that can be combined to support our game.

```
with Ada.Containers.Indefinite_Multiway_Trees ;;
with Ada.Containers.Indefinite_Vectors ;
with Interfaces ; use Interfaces ;

package words is

   package DictPkg is new Indefinite_Multiway_Trees(string) ;

   subtype AnagramsCount is integer range 1..64;
   package WordsPkg is new Indefinite_Vectors( AnagramsCount , string ) ;
   function EqualVector (Left,Right : WordsPkg.Vector) return boolean ;
   package SigTablePkg is new
```

```
          Indefinite_Ordered_Maps(Unsigned_64,WordsPkg.Vector,
                                  Interfaces."<" , EqualVector );

   type Dictionary is
      record
         tree : DictPkg.Tree ;
         sigwords : SigTablePkg.Map ;
      end record ;

end words ;
```

In the above specification, a **Dictionary** contains a **multiway tree** of all the words. This enables a quick verification that a given string is indeed a dictionary word. More interesting part of the **Dictionary** is the map from a signature value and a list of words that have the same signature. Then in order to compute the anagrams of a given word, we simply compute its signature and lookup the **sigwords** for the **Vector** of words.

2.3 Scrabblet

With the building blocks outlined above, this projectlet constructs the game in two modes; the computer either deals the hand and lets the user guess or in the player mode, the computer enumerates the words.

2.3.1 Dealer

The dealer mode is easily constructed to follow the steps:

- Generate a random sequence of the right number of letters called **the hand**.
- Generate all subsets of the hand and the dictionary looked up to confirm it is a word or it has anagrams.
- Keep a count of potential words
- If there are enough potential words, the game player is invited to guess words - keeping a tally of their score.

2.3.2 Player

In the player mode, the user provides **the hand** and by following the same idea the scrabblet game provides a list of words and computes a total score.

2.4 Implementation

2.4.1 Example - scores and signatures of words

```
 $ obj/score etc/dictionary.txt CREEPIE
     161524 words loaded
Word > CREEPIE
Score          11 Signature              4089745
Word > CRAP
Score           8 Signature                32330
Word > TARP
Score           6 Signature               459086
Word > TRAP
Score           6 Signature               459086
Word > TRAPZOID
It is not a word
Word > TRAPEZOID
Score          21 Signature        3859507079582
Word > TWITTER
Score          10 Signature             90946669
Word > SCRABBLE
Score          14 Signature             49902270
Word > SCRUBBLE
It is not a word
Word > SCROBBLE
It is not a word
Word > SCRIBBLE
Score          14 Signature            573876105
Word >
```

2.4.2 Example - enumerate anagrams

```
 obj/anagrams.exe etc/dictionary.txt CRPEI
     161524 words loaded
ListAnagrams: CRPEI              4089745
CREEPIE
CREEPIER
CREPIER
CRIPE
EPEIRIC
PIECER
PIERCE
PIERCER
PRECIPE
PRECIPICE
PREPRICE
```

```
PRICE
PRICER
PRICIER
RECIPE
REPRICE
```

2.4.3 Example - anagrams of subsets

```
 obj/main etc/dictionary.txt
     161524 words loaded
 128
Sub word A ———————————————
Sub word B ———————————————
 ...
 ...
 ...
Sub word ABLE ———————————————
ABELE
ABLE
BABBLE
BABEL
BALE
BLAE
LABEL
LABELABLE
LABELLA
          14 is a possible word. Signature:          2442 ABLE
Sub word T ———————————————
Sub word AT ———————————————
```

2.4.4 Example - dealer generation of the hand

```
obj/gamesetup
IAAJOGA Regenerate: Vowel Count  5
HIIOLNA Regenerate: Vowel Count  4
UIKALWD
GEUITVP
SWDSRIX
EISAIIO Regenerate: Vowel Count  6
 HETVTB
TAUBWTE
GUIHAJA Regenerate: Vowel Count  4
QIVAEPE Regenerate: Vowel Count  4
RUNESEI Regenerate: Vowel Count  4
RNAEFPS
```

PTADRNI

In the above example, we count the number of vowels and if there are too many (¿3) it may be considered for rejection and a new one generated.

2.4.5 Example of a game - dealer mode

```
obj/scrabblet -d etc/dictionary.txt dealer
     161524 words loaded
Dealer Mode
LEOAMPA Regenerate: Vowel Count  4
ZDAIOAT Regenerate: Vowel Count  4
ELIOTAP Regenerate: Vowel Count  4
 128
Your hand is RUOLFNT you have at least 25 words you can construct
Word > front
front is a word
Word > loot
loot is a word
Word > loft
loft is a word
Word > troll
troll is a word
Word > score
         24
Word > turf
turf is a word
Word > turn
turn is a word
Word > trunt
Not a word trunt
It is not a word

Word > show
front
loot
loft
troll
turf
turn
Word > hand
RUOLFNT
Word > plot
plot is a word
plot is not formed correctly. Please use only letters dealt
```

```
Word > flout
flout is a word
Word > score
          43
Word > floor
floor is a word
Word > front
front is a word
You have already guessed that word front
Word > loft
loft is a word
You have already guessed that word loft
Word >
Not a word
It is not a word
Word > runt
runt is a word
Word > score
          55
```

2.4.6 Example - player mode

Surprising how interesting a sequence like **abcdef** can be in this context.

```
obj/scrabblet -d etc/dictionary.txt player abcdef
     161524 words loaded
Player mode
 64
ABBA       score          8
BABA       score          8
CACA       score          8
ABACA      score          9
BACCA      score         11
DADA       score          6
BEEBEE     score         10
ABBE       score          8
BABE       score          8
CAECA      score          9
CECA       score          8
BACCAE     score         12
...
...
```

```
CABBED     score      13
BAFF       score      12
CAFF       score      12
DAFF       score      11
BEEF       score       9
FEEB       score       9
CAFE       score       9
EFFACE     score      14
FACE       score       9
FEED       score       8
DAFFED     score      14
DEAF       score       8
FADE       score       8
FADED      score      10
BEEFED     score      12
BAFFED     score      15
DECAF      score      11
DEFACE     score      12
DEFACED    score      14
EFFACED    score      16
FACADE     score      12
FACED      score      11
```

2.5 Ada Implementation

```
https://gitlab.com/ada23/words.git
```

3 References

Programming for the Puzzled by **Srini Devadas** (`https://mitpress.mit.edu/books/programming-puzzled`)